A SUPPORT SYSTEM FOR ERROR CORRECTION QUESTIONS IN PROGRAMMING EDUCATION

Yoshinari Hachisu and Atsushi Yoshida
Faculty of Science and Engineering, Nanzan University, 27 Seirei-cho, Seto-shi, Aichi-ken, Japan

ABSTRACT

For supporting the education of debugging skills, we propose a system for generating error correction questions of programs and checking the correctness. The system generates HTML files for answering questions and CGI programs for checking answers. Learners read and answer questions on Web browsers. For management of error injection, we have analyzed types of errors and defined the processes of error injection as code transformation patterns. The system synthesizes code fragments including errors by transforming correct code fragments according to the selected patterns. Full coverage of all possible answers is difficult. Instead, we have adopted a strategy to restrict editable points and possible answers from the educational view. To confirm the effectiveness of the system, we have generated questions using several examples and applied them to a programming exercise as an evaluation experiment.

KEYWORDS

Error Correction Questions, Programming Education, Debugging

1. INTRODUCTION

In programming courses, students learn skills for coding, code reading, and debugging through various kinds of exercises. Although debugging is important for actual software development, it is difficult for students, even who have a good understanding of programming, to acquire skills to debug programs effectively [6][7]. In this paper, we focus on learning support for debugging, whose controlled exercises are difficult to be provided.

Typical exercises in programming courses are describing full codes and filling empty boxes embedded in the texts of codes. Through describing full codes, learners find errors and inevitably try to debug for detecting faults. Debugging experiences, however, are different for each learner, and teachers cannot control experiences for them to use all necessary debugging skills. In the case of exercises using questions for filling empty boxes, learners may not have chances to debug because the codes they read do not include any errors, and they can fill answers without compiling and testing. For controlling their experiences of debugging, exercises using error correction questions are suitable. Learners read codes and need to detect errors that are injected purposely. Teachers can select errors for injecting according to debugging skills to be learned.

Preparing error correction questions is, however, a tough work for teachers. It requires injecting various errors in code fragments and check the correctness of all possible answers, some of which may be unexpected one but correct. There are some researches for supporting these work flows [2][7][8][10], but a few ones for automatic systems of question generation and correctness checking [2][8]. The key questions about constructing the systems are how to manage error injection into code fragments and how to cover all possible answers in correctness checking.

In this paper, we propose a system for generating error correction questions and checking the correctness. The system generates HTML files including editable text forms for answering questions and CGI programs for checking answers. They are deployed on a Web server and learners read and answer questions on Web browsers. For management of error injection, we have analyzed types of errors and defined the processes of error injection as code transformation patterns, which we call *error patterns*. The system synthesizes code fragments including errors by transforming correct code fragments according to selected error patterns. If the system allows the learners to edit any points in codes, it needs to accept all possible correct answers. Full coverage of all possible answers is difficult. Instead, we adopt a strategy to restrict possible answers to the

ones that is reasonable for learning objectives, which mean the syntax and semantics of the language, typical program descriptions, and algorithms that learners should understand. We have analyzed types of possible answers and propose constraints of editable points in codes on answering questions. For evaluating the system, we have implemented a prototype of the system and generated questions for learning the syntax and semantics of conditional branches, loops, arrays, strings, pointers, and structures, and a sorting algorithm.

The main contribution of this paper is to provide a fundamental framework of automatic generation and checking answers of error correction questions for debugging exercises. On the framework, the errors are defined as code transformation patterns, and this makes it easy for teachers to add new programs and errors. We hope that the discussion about what errors and programs are effective in terms of acquiring debugging skills and how we collect them in practical programming exercises is opened up.

In the following, we analyze error correction questions and discuss reasonable constraints in Section 2, and show a design of the system in Section 3. In Section 4, we show an experiment for evaluating the system and discuss its results. We show related works in Section 5 and conclude our research in Section 6. Though we use the C language in this paper, our study is not restricted to it.

2. TYPES OF ERROR CORRECTION QUESTIONS

2.1 Error Correction Questions

```
 1: #include <stdio.h>
 2: #define MAXSIZE 128
 3:
 4: int main(void)
 5: {
 6:   int data[MAXSIZE];
 7:   int size, sum, count, i;
 8:   double avg;
 9:
10:   sum = 0;
11:   size = 0;
12:   while (scanf("%d", &data[size])
13:                != EOF) {
14:      sum += data[size];
15:      size++;
16:   }
17:
18:   avg =  (double) sum / size;
19:   printf("avg = %f\n", avg);
20:
21:   count = 0;
22:   for (i = 0; i < size; i++) {
23:      if (data[i] > avg) {
24:         count++;
25:      }
26:   }
27:   printf("greater than avg: %d\n",
28:          count);
29:
30:   return 0;
31: }
```

(a) Correct Program

```
 1: #include <stdio.h>
 2: #define MAXSIZE 128
 3:
 4: int main(void)
 5: {
 6:   int data[MAXSIZE];
 7:   int size, sum, count, i;
 8:   double avg;
 9:
10:   sum = 0;
11:   size = 0;
12:   while (scan("%d", _data[size])
13:               != EOF) {
14:      sum += data[size];
15:      size++;
16:   }
17:
18:   avg =  (double) sum / size;
19:   printf("avg = %f\n", _avg);
20:
21:   for (i = 1; i <= size; i++) {
22:      if (data[i] > avg) {
23:         count++;
24:      }
25:   }
26:   printf("greater than avg: %d\n",
27:          _count);
28:
29:   return 0;
30: }
```

(b) Program including Errors

Figure 1. Error correction question (1): condition, loop, and array

An error correction question contains a program code including errors and requires the learners to modify them correctly. An example is Fig. 1, whose specification is "Calculate the average value while reading a list of integer values in a file, and then print a number of values that are greater than the average." Program (a) in Fig. 1 is a correct program, and program (b) includes four errors: in Line 12, (1) the function name **scan** is a misspell of **scanf**, and (2) an address operator **&** is missing in an argument; in Line 21, (3) the initializer and the condition in the **for** loop do not match to the range of the array referred in the loop; in Line 23, (4) variable **count** is referred without initialized. The underlined italic texts are editable elements, that is, learners can modify them. The blank lines at Line 9, 17, 20, and 28 are also editable, where learners can add statements. When a question has many editable elements, it tends to become difficult and to allow unexpected answers. In Section 2.5, we discuss the restriction of editable code elements from the educational view. Fig. 1 is a question designed for beginners of programming who is learning the syntax and semantics of conditional branches, loops, and arrays. By this question, we hope that learners learn the usage of the **scanf** function, necessity of variable initialization, and a typical loop to scan an array.

```
 1: struct person {
 2:     char name[64];
 3:     double height, weight;
 4: };
 5:
 6: void swap(double *a, double *b)
 7: {
 8:     double _tmp;
 9:     tmp = *a; *a = *b; *b = tmp;
10: }
11:
```

```
12: void sort_by_height(
13:       struct person p[], int size)
14: {
15:   int min, i, j;
16:
17:   for (i = 0; i < size-1; i++) {
18:     min = i;
19:     for (j = i+1; j < size-1; j++) {
20:       if (p[j].heihgt < p[min].height ) {
21:             min = j;
22:           }
23:       }
24:       swap(_p[j].height, _p[min].height);
25:     }
26: }
```

Figure 2. Error correction question (2): a sort program of an array of structures

We show another example in Fig. 2. It is a more complicated program using pointers, structures, and the selection sorting algorithm; the structure **person** represents a personal physical datum for each person and the function **sort_by_height** sorts an array of the structure. The sort function gets an array by argument **p** and its size by argument **size**, and it sorts **p** in ascending order of a member **height** with the selection sort algorithm. It contains five errors: (1) in Line 6 and 8, the type of function **swap**'s arguments and local variable **tmp** must be **struct person**; (2) in Line 19, the condition must be **j < size**; in Line 24, (3) all address operator **&**s are missing for both arguments; (4) the index **j** of array **p** must be **i**; (5) all member references of **height** are unnecessary.

These examples imply that error correction questions are more difficult than fill-in-the-blank questions. Fig. 3 shows (a) an error correction question and (b) a fill-in-the-blank one about **if**-branches and strings. The function **strconv** changes upper letters in the ASCII string **s** to lower ones and lower letters in **s** to upper ones; for example, **"Hello World!"** is changed to **"hELLO wORLD!"**. Program (a) includes three errors: in Line 3, (1) the equivalence operator in the condition of the **while** is wrong; (2) the logical operators in Line 4 and 7 are wrong; in Line 7, (3) **else** is missing before **if**. In our experiences, error (3) is relatively hard to find at a glance, although this program is simple and short. On an error correction question, firstly, learners read a program, secondly, they find errors, and finally they correct the codes, that is, adding, deleting, replacing, or moving codes. Finding and correcting errors require understanding the control flows, data flows, and logics of programs in addition to the syntax of the language. On the other hand, program (b) shows explicitly blank boxes where learners need to write correct codes. They may write correct codes without comprehension of programs. For example, at the first blank in Line 7, there are clear hints, **if** in Line 4 and the following parentheses. Furthermore, from the syntactical correctness, it should be filled with either **if**, **else if**, **while**, or **switch**.

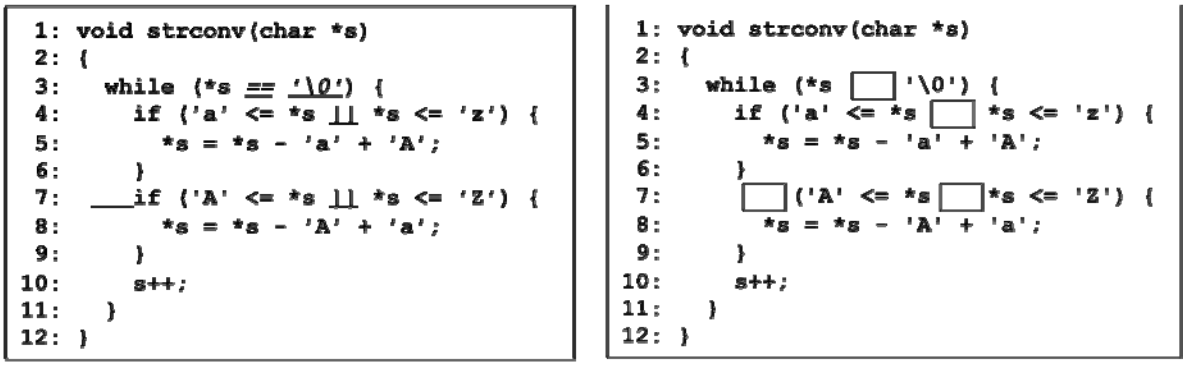

Figure 3. Error correction question and fill-in-the-blank question: **`if`**-branches and strings

When we generate error correction questions, we need to maintain the readability of codes. Codes should have comprehensive structures of the units of inputs, main processes, and outputs because learners may have less skill for reading. Unnatural descriptions in codes may give undesirable hints to learners. We have adopted the following coding styles: (S1) a variable is used for one purpose; (S2) the initialization of a variable that is used in a loop should locate just before the loop; (S3) two semantic process units are separated by a blank line; (S4) no successive blank lines are allowed. While injecting errors, removing statements may cause two successive blank lines in codes. They should be integrated into one blank line because they indicate a definite lack of a statement. For example, in Fig.1 (a), when we remove the assignment statement in Line 21, we also remove the following new line because Line 20 is a blank line.

Table 1. Samples of errors for error correction questions of programs

Learning Objective	Error	Edit Operation	Correctable Point(s)	Possible Represen-tation(s)	Type
Reading primitive values with **`scanf`** function	Missing address operator **&**s before arguments (Fig. 1 error (2))	delete	one	one	(a)
Reading strings with **`scanf`** function	Unnecessary address operator **&**s before arguments (**`char []`**)	insert	one	one	(a)
Operators	Wrong operator (Fig. 3 error (1),(2))	replace	one	one	(a)
Branches	Missing else (Fig. 3 error (3))	delete	one	one	(a)
Loops	Wrong initialization and condition in a loop (Fig. 1 error (3))	replace	one	multiple	(b)
Types	Wrong type (Fig. 2 error (1))	replace	one	one	(a)
Structures	Unnecessary member references (Fig. 2 error (5))	insert	one	one	(a)
Data flows of variables and initialization	Using a variable without initialization (Fig. 1 error (4))	delete	multiple	one	(c)
Data flows in a loop	Moving the assignment before a loop inside the loop	move	multiple	one	(c)
The elements to be swapped in the selection sort algorithm	Using wrong indexes on swapping elements in an array. (Fig. 2 error (4))	replace	one	one	(a)

2.2 Errors for Questions

For analyzing types of errors used in error correction questions, we have collected errors in our programming courses and the related papers [3][6][9][11]. These papers have reported that the students often make syntactical mistakes such as missing a semicolon and lacking one of a pair of curly braces or parentheses. In general, it is relatively easy for students to correct syntactical errors because a compiler detects them and shows error messages with line numbers. We have collected logical errors such as using a variable without initializing or initialized with a wrong value because these errors significantly contribute the development of debugging skills Compilers do not detect these types of errors, and learners need to read a program carefully and understand its logic.

We show examples of errors and the learning objectives of them in Table 1. We have analyzed these errors in terms of edit operations and the numbers of correctable points and possible representation. A correctable point means an editable point at which we can write a possible representation. A possible representation is one of the code fragments that are acceptable as correct answers. It may be semantically different from the original (see Section 2.4).

2.3 Edit Operations for Injecting Errors

We have analyzed edit operations to inject errors in code fragments. We have classified the operations into four types: *insert*, *delete*, *replace*, and *move*. The column *edit operation* in Table 1 shows operations for each error. Though an operation *move* is a combination of *delete* and *insert*, we have identified it as a primitive because the two operations should be occurred simultaneously.

The errors injected by deleting code elements or replacing code elements with others are the most typical ones. A missing assignment of an initial value and a wrong relational operator in a loop condition are examples. They are caused by deleting the assignment or replacing the operator with another one. Error correction questions of theses types are relatively easy to prepare. The number of these errors we found is greater than the ones of others. Instead, the errors of *insert* and *move* are restricted. Though it is not difficult to insert or move an element in codes, these errors would happen rarely because they break data flows or introduce unnecessary statements in contexts, and it is easy for learners to find them.

If we may not consider all possible answers, checking the correctness of answers becomes easy to be implemented; the system needs only to check that each modified element is same with the original one. For the *move* operation, the system needs to check both of modifications of *insert* and *delete* are correct. The difficulty is to cover all possible answers, and we discuss it in the next section.

2.4 The Number of Correctable Points and Possible Representation

We have analyzed correctable points and possible code fragments for the errors that we found. Possible answers mean a set of answers whose positions in codes and representations are different. We ignore the differences of the representations of syntactical equivalent expressions, such as a difference of **`a+b`** and **`b+a`** because it is possible to generate all representations of expressions systematically. Our system is for beginners education, and the expressions of the answers are small, for which it is possible to generate all alternatives in practical time. We also ignore differences of white spaces. We explain how to ignore the differences in our implementation in Section 3.1. In the following, we consider the possible answers whose position and/or semantics are different.

The results of the analysis are shown in Table 1 as the numbers of correctable points and possible representations. For considering the difficulty of the correctness checking, we categorize errors into four types: (a) one correctable point and one representation, i.e., not existing any other answer, (b) one correctable point and multiple possible representations, (c) multiple correctable points and one representation, and (d) multiple correctable points and multiple possible representations.

Type (a) represents the errors for which the original positions and code fragments are the unique answers. The supporting system checks only one position and one fragment for each error and is easy to be implemented.

Type (b) represents the errors for which the original positions are unique answers, but there are variations in code fragments to be placed. For example, an *n*-times loop including an error, `for (i=1; i<n; i++)` can be corrected as either `for (i=0; i<n; i++)` or `for (i=1; i<=n; i++)`. They cannot be checked syntactically, and this correctness depends on contexts.

Type (c) represents the errors for which the original code fragments are the unique answers, but the position of each error is not fixed to one. An error of this type occurs when a statement is removed. For example, in Fig. 1 (b), the initialization of variable `count` can be inserted at any lines between Line 9 and 20.

Type (d), which does not appear in Table 1, represents the errors that have multiple answers in positions and code fragments. This type of errors makes the system complicated. The answers other than the original ones tend to be unnatural and inadequate from the educational view. For example, we can correct the errors at Line 21 in Fig. 1 (b) by replacing `data[i]` with `data[i-1]` in Line 22. If we restrict the correctable points of the above example to Line 21, the error becomes type (a), which is the simplest type. This example suggests us that restriction of acceptable answers is a reasonable approach.

2.5 Restriction of Correctable Points and Editable Points

How to restrict correctable points is a difficult question because they depend on the semantics of errors and target codes. One of the simplest ways is to restrict to the original positions and code fragments, as type (a). The errors in type (b) can be changed to type (a) by restricting correctable points. For example, in an error code fragment of an *n*-times loop, `for (i=1;i<n;i++)`, by allowing modification of the operator `<` in the condition, only `for (i=1;i<=n;i++)` can be acceptable. For accomplishing this, the system needs the ability to modify fine-grained elements.

Strong restrictions, however, provide clear hints to learners and easily lead them to the answers. To avoid correctable points becoming clear hints, we introduce editable points. Editable points exist at all positions similar to correctable points and the learners cannot distinguish them on the system. For example, in Fig. 1 (b), the correctable point of lacking operator `&`s is at the front of the second argument at Line 12, and the editable points are at the front of the second arguments in the `printf` functions at Line 19 and 27. Editable points include correctable points, and we can define them in the same way of correctable points. The difference between them is the modification of texts. When they are injected in the codes, the texts at the correctable points are replaced with error ones, but the ones at the other editable points are not changed. This introduction of editable points embraces the contradiction that they increase possible answers, which have been excluded by the restriction of correctable points. Unfortunately, this contradiction is not able to resolve systematically. How to select patterns for editable points are responsible for the users as teachers.

An error with multiple correctable points, like type (c), occurs when an assignment is deleted. The error can be corrected by adding the original assignment at any point unless it is to preserve the original data flows. Strictly speaking, the correctness checking of type (c) requires data flow analysis, which makes the system complicated. To keep the system simple, we introduce the constraints reasonable from the educational view: (i) only assignments for initialization can be deleted; (ii) editable points where learners can insert statements are restricted to blank lines. The rule (i) is introduced because missing initializations are typical errors, and the questions deleted other kinds of assignments become the same with fill-in-the-blank ones for understanding algorithms which are out of our targets. The rule (ii) prevents the learners from confusing by multiple possible answers. Under these constraints and the coding style (S1) and (S2) in Section 2.1, we can identify the valid correctable points without data flow analysis, which are blank lines located between sibling statements before the deleted assignment. For example, in Fig. 1 (b), the correctable points are the blank lines at Line 9, 17, and 20; the one at Line 28 is not a correctable point.

The errors of type (d) are problematic, but they are minor in our experiences. The purpose of the error correction questions is to develop skills for identifying typical errors. The errors of type (d) deeply depend on semantics of code, and they are not typical. We should avoid this type of errors or try to change it to type (c) or (a) by restriction of editable points. In this paper, we do not discuss this anymore.

3. AUTOMATIC GENERATION OF ERROR CORRECTION QUESTIONS

3.1 Overview of the Supporting System

We have developed a supporting system of error correction questions. Fig. 4 shows an overview of the system. The system consists of two components. One is a presentation system that provides a set of error correction questions to the learners, and they answer on it. The answers are checked by the system, and the results are returned to the learners. It is implemented as a Web system, i.e., a set of HTML files, JavaScript programs and Perl CGI scripts that are deployed on a web server. The learners read and answer questions on web browsers. The other component of the system is a generating system of error correction questions. A user as a teacher selects a correct program and error patterns. The system applies the patterns to the correct program and generates both of an HTML file of the question and a CGI program for checking answers. The HTML file contains a program including injected errors and editable points.

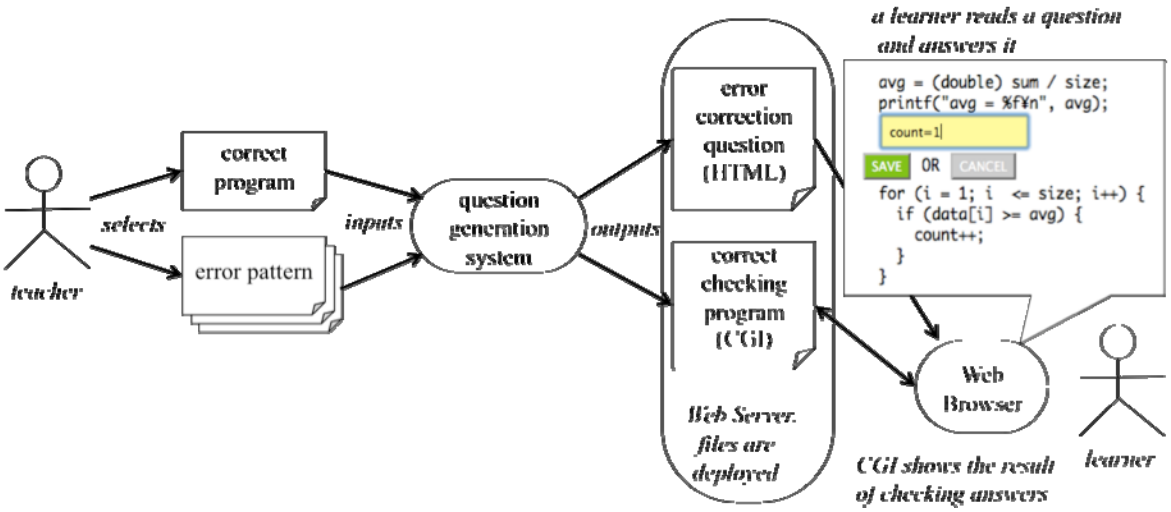

Figure 4. An overview of the supporting system for error correction questions.

The checking CGI programs can accept variants of expressions, such as a variant `b+a` for `a+b`. The equivalence of expressions is complicated to evaluate in practice, and the development of the evaluation algorithm is out of the scope of this paper. We have implemented an evaluation code optimized for our examples. Answers for all examples are small expressions, and we do not accept unnatural answers, such as `a+0` for `a` and `5-5` for `0`. Therefore, the number of variants that we should consider is small.

The presentation system needs to hide editable points from the learners. If all editable points explicitly appear on question pages, the learners can easily guess the answers. For this requirement, we have adopted JEIP[1], a plug-in of jQuery, which hides editable points from a learner until the user moves the mouse over the texts on them. If the user clicks an editable point, an input field and a save button appear at the point. After changing and saving all editable points where the user consider need to modify, the user can submit answers to the system by pressing the submit button.

A demonstration of error correction questions is available at http://ecq.tebasaki.jp/.

[1] jQuery Edit In Place (JEIP), http://josephscott.org/code/javascript/jquery-edit-in-place/

ISBN: 978-989-8704-08-5 © 2014

3.2 Error Patterns

For managing correct programs and errors separately, we define errors as code transformation patterns. We also define editable points in the same way of errors, which do not modify programs in practice. This separation makes it easy to add a code fragment and another type of errors newly. We have implemented the system on a program transformation, called TEBA [4], which has a parser of code fragments including additional symbols, such as program patterns. TEBA also provides a transformation system on token-based syntax trees, and it allows modifying the fine-grained elements as discussed in Section 2.5.

An error pattern consists of two parts, *before-part* and *after-part*. A before-part is a target code to be modified, and an after-part is a new code as a replacement. For abstraction of syntactic elements, typed pattern variables can be used. In error patterns, the correct tokens in the before-part and the error tokens in the after-part are surrounded by the special tag, **<@** and **@>**. In an error patterns of *delete* type, the tag in after-part surrounds an empty token since error tokens never exist. The line where the tag exists becomes a blank line after the error pattern is applied. In an error pattern of *insert* type, the correct tokens in the before-part are empty. In one of *move* type, we use the tag, **<#@** and **@>**, for grouping a set of deleted tokens and inserted ones. We describe an injection of editable points as an error pattern whose before-part and after-part are the same. We show three examples of error patterns in Fig. 5.

```
% before
for ( ${e1:EXPR} = <@ 0 @>; ${e21:EXPR} <@ < @> ${e2r:EXPR}; ${e3:EXPR})
   ${s:STMT*} $;
% after
for ( $e1 = <@ 1 @>; ${e21} <@ <= @> ${e2r}; ${e3})
   ${s} $;
% end
```

(a) replacing initializer and the operator of the condition in a `for` loop (Fig. 1 error (3))

```
% before
<@ ${var:ID_FVAR} = ${lit:LITERAL}; @>
for (${e1:EXPR}; ${e2:EXPR}; ${e3:EXPR} )
    ${s:STMT*} $;
% after
<@ @>
for (${e1}; ${e2}; ${e3} )
  ${s} $;
% end
```

(b) deleting an assignment just before a `for` statement (Fig. 1 error (4))

```
%before
swap(${a1:EXPR}<@ @>, ${a2:EXPR}<@ @>);
%after
swap(${a1}<@ .height @>, ${a2}<@ .height @>);
%end
```

(c) inserting unnecessary member references `.height` (Fig 2. error(5))

Figure 5. Error patterns as code transformation

4. DISCUSSION

4.1 Generality of Error Patterns

To confirm that the system can generate error correction questions, we have defined 15 patterns for injecting errors and 10 patterns for setting editable points, and applied them to 4 sample programs: Fig. 1, Fig. 2, Fig. 3, and a binary search function. The error patterns we defined are varied in generality. For example, a pattern of removing addressing operators **&**s can be applied to all code fragments that include the operators. The error pattern Fig. 5 (c) is less general, which depends on the function **`swap`** and the structure **`person`**. In our defined patterns, 10 of 25 patterns are program specific. These patterns are difficult to be reused, but they are useful for generating questions optimized for learning objectives. For example, in Fig 2., while we have described an error pattern for inserting unnecessary references (Fig. 5 (c)), we have selected a member name, **`height`**, because the specification of the program requires to sort data by the member. We have set an

editable point on the indexes of **p[j]**, but not the ones of **p[min]**. If both of them are editable, another answer becomes acceptable; the answers, **p[min].height > p[j].height** in Line 20 and **swap(&p[min], &p[j])** in Line 24, are valid. We also have made a space editable before variable **tmp** in Line 8 because we expect that learners may insert ***** in the same way of the parameters ***a** and ***b** without thinking semantics deeply.

From our experiences, the error patterns for learning the syntax and semantics of the language tend to be general, and the ones for learning algorithms tend to be specific. For improving the reusability, we need to investigate the way to make program specific patterns more general. It may be possible to describe the contexts of applying patterns and its edit operations separately. Though the contexts depend on programs, we expect operations can be described in general styles.

4.2 Experiment and Evaluation

We also have made an experiment of the programming exercise using the generated error correction questions. The subjects are 10 undergraduate students in the third grade, who had taken programming courses and have the skills for developing small programs. The purpose of the experiment is to find unexpected answers. As a result, we have not found unexpected answers that are correct. This means the restriction of our approach works as expected. We, however, have found redundant answers. For example, a student has inserted **avg = 0;** at Line 9 in Fig. 1 (b). The variable **avg** is never referred before the assignment at Line 18, and no initialization is needed. In the question of Fig. 2, there was an answer that the condition of the **for** at Line 17 was changed to **i < size**. Though it executes the redundant process on **i == size-1**, it produces the same result as the original program. How to treat redundant answers depends on the purpose of the exercises. Our system cannot support the case that the teachers want to judge redundant answers as correct because the correctness checking is implemented based on text matching. For checking redundant answers, it may be effective to compare the results of the original codes and the answered codes by tests for the sufficient coverage of inputs, while it makes the system more complicated and requires more efforts for teachers to prepare adequate test sets.

5. RELATED WORKS

AEGIS[2] is a system to generate questions from XML documents and supports three types: multiple choice, fill-in-the-blank, and error correction. An error correction question is generated from a multiple choice question by fusing the choices to the code. AEGIS requires describing a full XML description of questions, including codes and errors, and does not support automatic synthesis of questions. Our system separates descriptions of codes and error patterns and synthesizes questions from them. The interface of error correction questions generated by AEGIS is quite simple. It shows the text of a code on which errors are marked, and the learner inputs answers in the text fields. The interface of our system hides errors and allows modifying the elements that are not in error. Our system provides an exercise environment similar to the traditional ones using papers and pens.

Itoh et al. proposed a method for generating error-correction exercises for learning algorithms [10]. It determines the fault positions by the algorithm design paradigm and injects faults by the syntax-directed faults patterns, which are specific to the algorithm education. By specifying a set of correct programs, an algorithm design paradigm such as divide-and-conquer, the number of errors to be injected, and the number of source code files to be generated, the system generates source codes including errors automatically. Though it supports error injection, it does not propose a method for the correctness checking. Our system allows injecting an error at any positions, and it does not depend on specific domains of program educations. However, it requires manual selection of error patterns while considering possible answers. A support method for selecting patterns is a future work. Patterns should be selected from the multiple views such as learning objectives, the difficulty of questions, the degree of learners' understanding, and so on.

From the view of code transformation, our system is a kind of a mutation system [1][5]. A mutation system generates multiple variants of a code by adding small changes to its copies. The distinctive application is the test set evaluation in which test sets are tested how many variants they can detect as errors. Mutant systems add changes randomly, but our system adds changes in a restricted manner.

6. CONCLUSION

We have proposed a support system for generating error correction questions and checking answers. On the system, we can describe error patterns of injecting errors to codes, and the system synthesizes questions from them. The system also generates CGI programs for checking answers. We have collected some typical errors in programming and generated questions. We have showed the validity of our system by a small experiment. We need to make experiments with a large number of subjects.

Collecting other errors that are effective for acquiring debugging skills is a future work. Although some researches have reported syntax errors by novice programmers [3][6]][9][11], a few logical errors are known. While we can collect syntax errors by logging compile errors, to collect logical ones we need to investigate programming processes; we have an interest in how they correct faulty programs, for which compilers report no error. Work-in-progress codes and input data they select to test programs may be helpful for teachers to analyze their errors. An integrated development environment for education is one of the suitable systems to preserve learners' processes.

We hope that discussion about learning support for debugging using error correction questions is opened up.

ACKNOWLEDGEMENT

This work has partially been supported by Nanzan University Pache Research Subsidy I-A-2.

REFERENCES

Journal

[1] Jia, Y. and Harman, M., 2011, An Analysis and Survey of the Development of Mutation Testing, *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp.649-678.

[2] Suganuma, A. et al., 2005, Automatic Exercise Generating System That Dynamically Evaluates both Students' and Questions' Levels (in Japanese), *Journal of Information Processing Society of Japan*, Vol. 46, No. 7, pp. 1810-1818

[3] Tuugalei, I. and Mow, C., 2012, Analyses of Student Programming Errors In Java Programming Courses, *Journal of Emerging Trends in Computing and Information Sciences,* Vol. 3, No. 5, pp. 739-749.

[4] Yoshida, A. et al., 2012, A Source Code Rewriting System based on Attributed Token Sequence (in Japanese), *Journal of Information Processing Society of Japan*, Vol. 53, No. 7, pp. 1832-1849.

Conference paper or contributed volume

[5] Agrawal, H. et al., 1984, Design of Mutant Operators for the C Programming Language, *Technical Report SERC-TR41-P*, Purdue University.

[6] Ahmadzadeh, M., et al. 2005, An Analysis of Patterns of Debugging Among Novice Computer Science Students, *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Lisbon, Portugal, pp. 84-88.

[7] Chmiel, R. and Loui, M.C., 2004, Debugging: from Novice to Expert, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Virginia, USA, pp. 17-21.

[8] Hachisu, Y. and Yoshida, A., 2013, Generation of Error Correction Questions for Beginners of Programming (in Japanese), *Foundation of Software Engineering XX (FOSE2013)*, Kaga, Japan, pp.35-40.

[9] Hristova, M., et al., 2003, Identifying and correcting Java Programming Errors for Introductory Computer Science Students, *Proceedings of the 34th SIGCSE technical symposium on Computer Science Education*, Reno, USA, pp. 153-156.

[10] Itoh, R. et al., 2007, A Fault Injection Method for Generating Error-correction Exercises in Algorithm Learning, *Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training*, Kumamoto, Japan, pp. 200-205.

[11] Jackson, J., et al., 2005, Identifying Top Java Errors for Novice Programmers, *Proceedings Frontiers in Education 35th Annual Conference*, Indiana, USA. pp. T4C-24 - T4C-27.